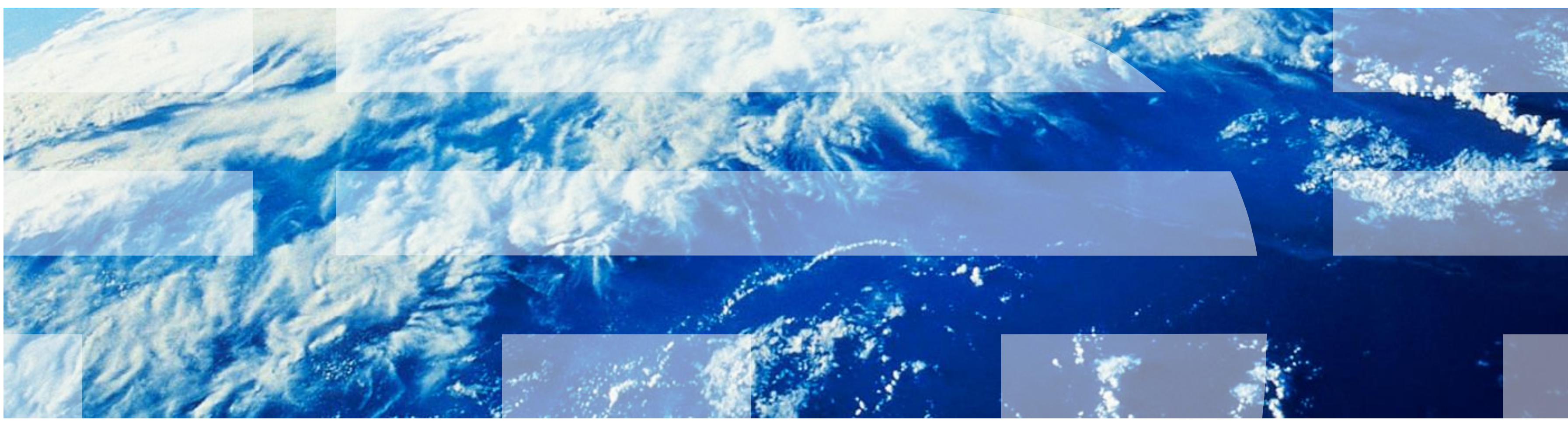
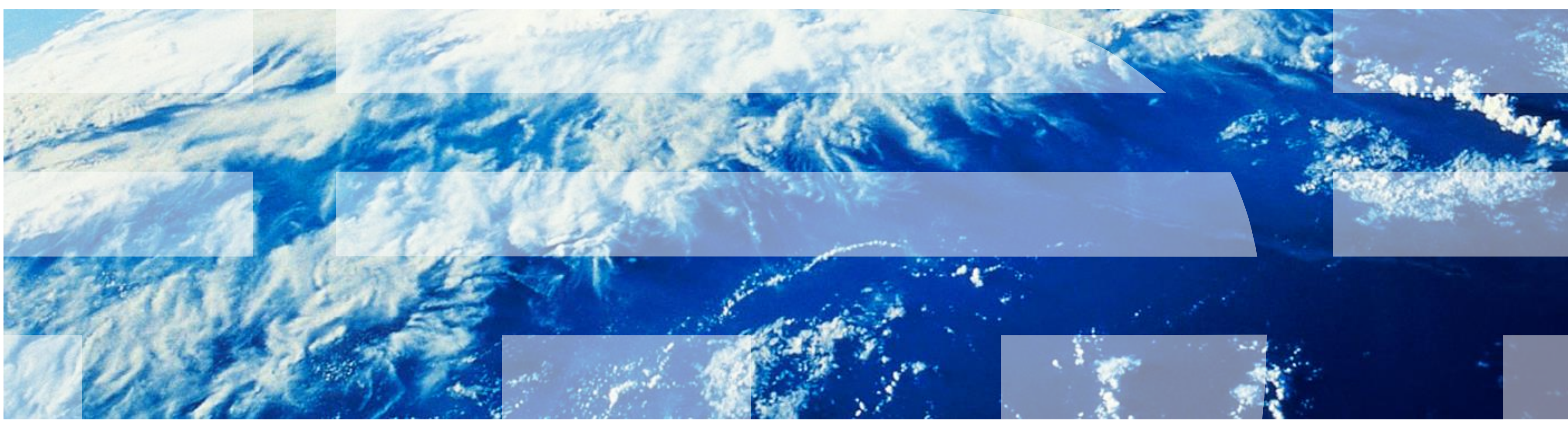
Borrowed from Shiva Shivakumar

Computer Systems for Data Science
Topic 3

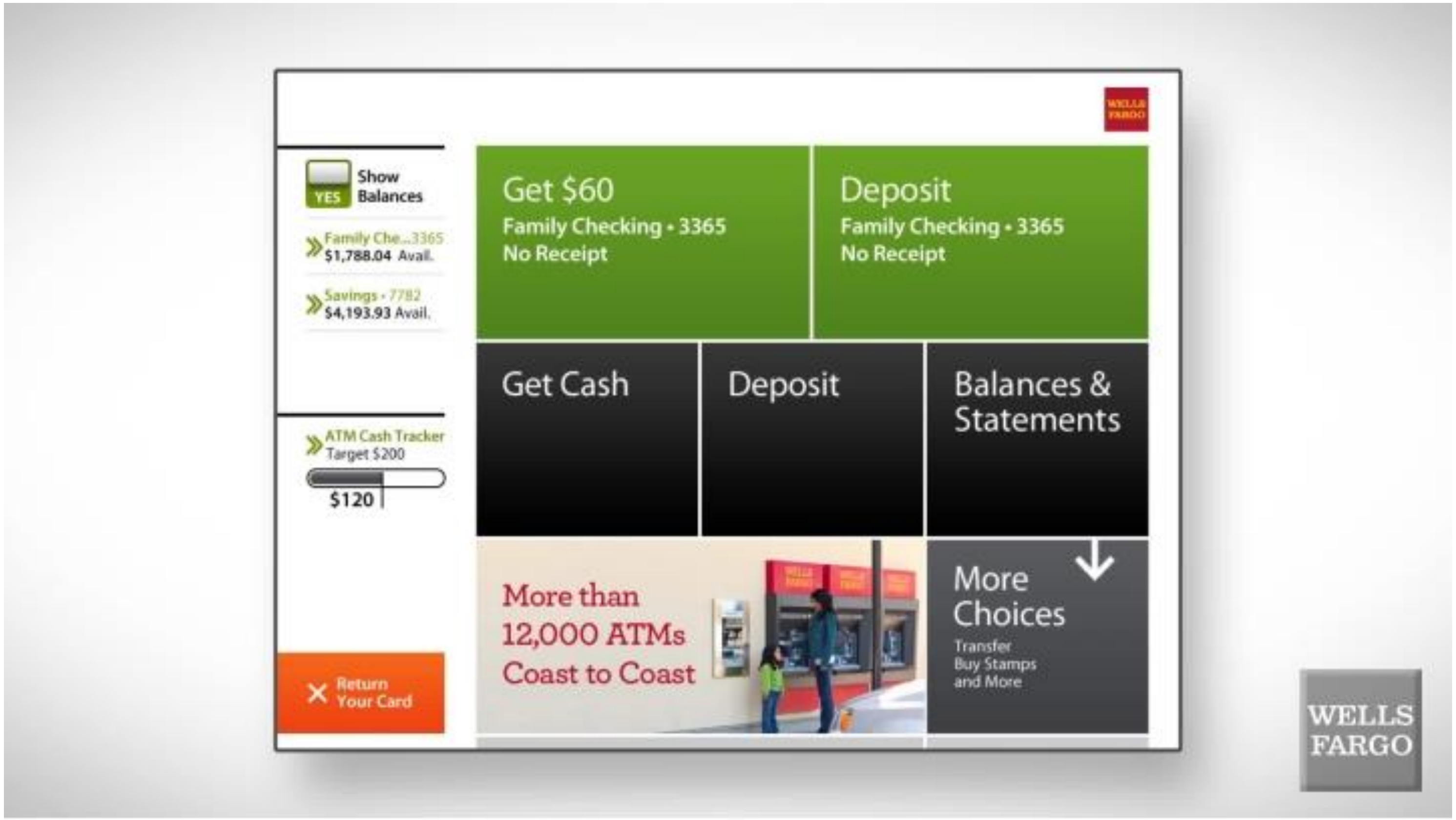
Transactions



Transactions



Motivating example: an ATM



Read Balance Give money Update Balance	vs	Read Balance Update Balance Give money

It’s not just about having the **correct** balance…

Visa does > 60,000 TXNs/sec with users & merchants

Want your 4$ Starbucks transaction to wait for a 10k$ bet in Las Vegas ?
(Transactions can (1) be quick or take a long time, (2) unrelated to you)

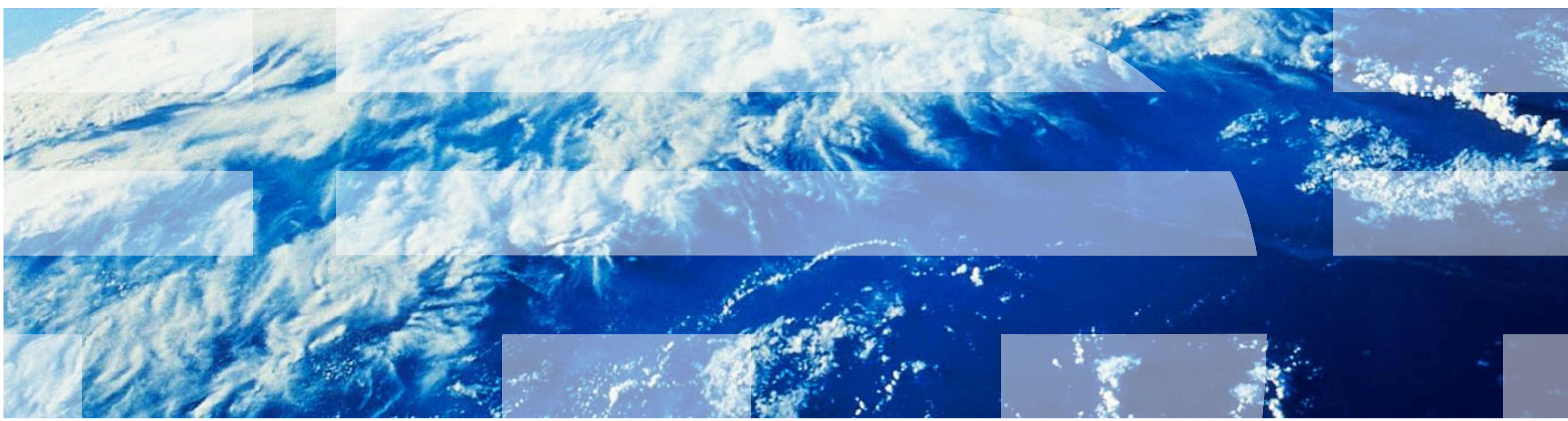
Transactions are not just used for finance

Transactions are at the core of

-- payment, stock market, banks, ticketing

-- Gmail, Google Docs (e.g., multiple people editing)

Transactions



Example: monthly bank interest transaction

Money

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance ($)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...		
30108		-110
40008		110
50002		22

'T-Monthly-423'

Monthly Interest 10%
4:28 am Starts run on 10M bank accounts
Takes 24 hours to run

```
UPDATE Money
SET Balance = Balance * 1.1
```

Example: monthly bank interest transaction **with crash**

Money

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

Money (@10:45 am)

Account	....	Balance ($)	
3001		550	??
4001		110	
5001		22	
6001		66	
3002		88	
4002		-200	??
5002		320	??
...			
30108		-110	
40008		110	
50002		22	??

‘T-Monthly-423’

Monthly Interest 10%
4:28 am Starts run on 10M bank accounts
Takes 24 hours to run
Network outage at 10:29 am,
System access at 10:45 am

Transactions: Basic Definition

A transaction (“TXN”) is a sequence of one or more *operations* (reads or writes) which reflects *a single real-world transition*.

TXN either happened completely or not at all

```
START TRANSACTION
        UPDATE Product
        SET Price = Price – 1.99
        WHERE pname = ‘Gizmo’
COMMIT
```

Transactions in SQL

- In “ad-hoc” SQL, each statement = one transaction

- In a program, multiple statements can be grouped together as a transaction

```
START TRANSACTION
        UPDATE Bank SET amount = amount – 100
        WHERE name = ‘Bob’
        UPDATE Bank SET amount = amount + 100
        WHERE name = ‘Joe’
COMMIT
```

Motivation for Transactions

Grouping user actions (reads & writes) into *transactions* helps with two goals:

1. **Recovery and Durability**: Keeping the DB data consistent and durable in the face of crashes, aborts, system shutdowns, etc.

2. **Concurrency**: Achieving better performance by parallelizing TXNs *without* creating anomalies

Motivation -- Recovery & Durability

1. **Recovery and durability** of user data is essential for reliable database (and other data science systems)

- The database may experience crashes (e.g. power outages, etc.)
- Individual TXNs may be aborted (e.g. by the user)

Idea: Make sure that TXNs are either **durably stored in full**, **or not at all**; keep log to be able to "roll-back" TXNs

Protection against crashes / aborts

Client 1:

```
INSERT INTO SmallProduct(name, price)
SELECT pname, price
FROM Product
WHERE price <= 0.99

DELETE Product
WHERE price <=0.99
```

Crash / abort!

What goes wrong?

Protection against crashes / aborts

```
Client 1:
          START TRANSACTION
          INSERT INTO SmallProduct(name, price)
                    SELECT pname, price
                    FROM Product
                    WHERE price <= 0.99

          DELETE Product
                    WHERE price <=0.99
          COMMIT OR ROLLBACK
```

Now we'd be fine! We'll see how / why this lecture

Motivation -- Concurrent execution

2. **Concurrent** execution of user programs is essential for good database performance.

- Disk accesses may be frequent and slow- optimize for throughput (# of TXNs), trade for latency (time for any one TXN)
- Users should still be able to execute TXNs as if in isolation and such that consistency is maintained

Idea: Have the database handle running several user TXNs concurrently, in order to keep throughput high

Multiple users: single statements

```
Client 1: UPDATE Product
                SET Price = Price – 1.99
                WHERE pname = ‘Gizmo’

Client 2:   UPDATE Product
                SET Price = Price*0.5
                WHERE pname = ‘Gizmo’
```

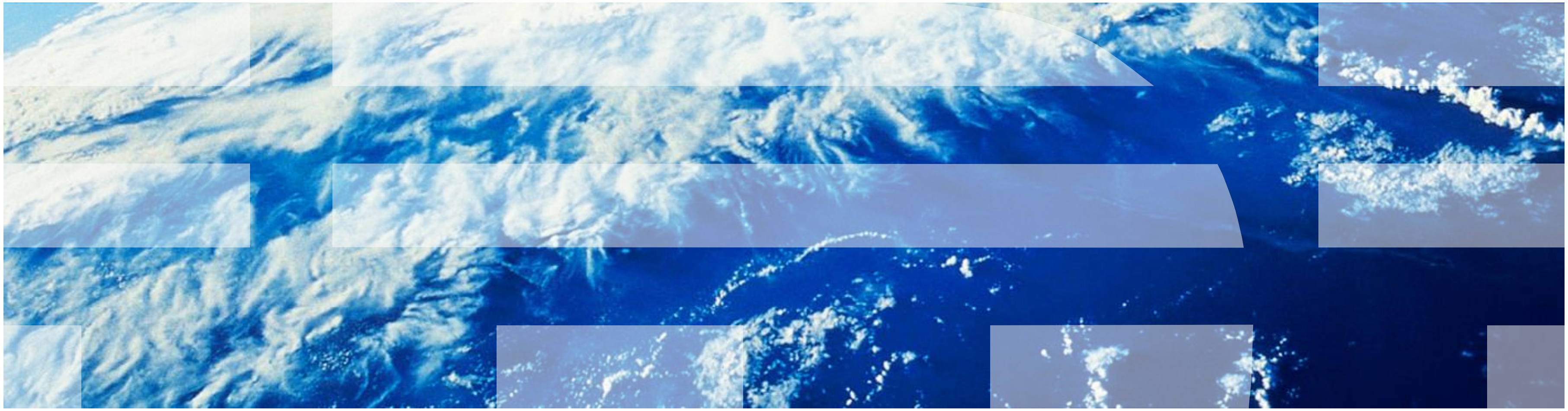
Two managers attempt to discount products *concurrently*- What could go wrong?

Multiple users: single statements

```
Client 1: START TRANSACTION
                              UPDATE Product
                              SET Price = Price – 1.99
                              WHERE pname = ‘Gizmo’
          COMMIT

Client 2: START TRANSACTION
                              UPDATE Product
                              SET Price = Price*0.5
                              WHERE pname=‘Gizmo’
     COMMIT
```

ACID
Atomicity, Consistency, Isolation, Durability



Transaction Properties: ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

ACID continues to be a source of great debate!

<u>A</u>CID: <u>A</u>tomicity

- TXN's activities are atomic: all or nothing
  - Intuitively: in the real world, a transaction is something that would either occur *completely* or *not at all*
- Two possible outcomes for a TXN
  - It *commits*: all the changes are made
  - It *aborts*: no changes are made

ACID: Consistency

- The tables must always satisfy user-specified *integrity constraints*
  - *Examples:*
    - Account number is unique
    - Stock amount can’t be negative
    - Sum of *debits* and of *credits* is 0

- How consistency is achieved:
  - Programmer writes a TXN to go from one consistent state to a consistent state
  - *System* makes sure that the TXN is atomic
  - → Assuming system maintaining atomicity, this is often the user’s responsibily

ACID: Isolation

- A transaction executes concurrently with other transactions
- **Isolation**: the effect is as if each transaction executes in *isolation* of the others.
    - E.g. Should not be able to observe changes from other transactions during the run

ACID: Durability

- The effect of a TXN must continue to exist (*"persist"*) after the TXN
  - And after the whole program has terminated
  - And even if there are power failures, crashes, etc.
  - And etc…

- Means: Write data to disk
  - And in data center settings: replicate data, backup, etc.

Challenges for ACID properties

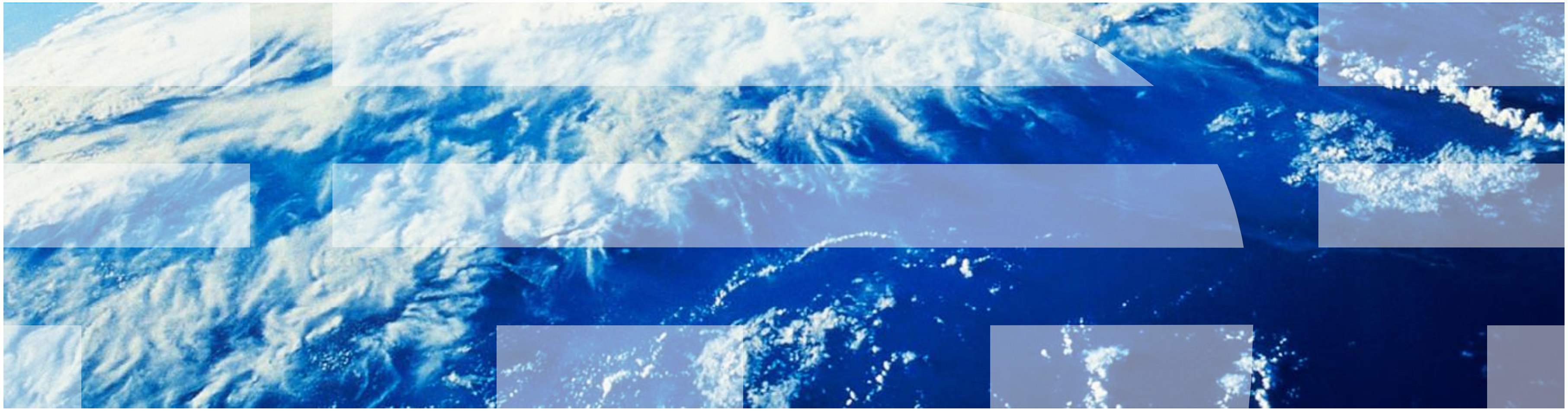
- In spite of power failures (i.e., in spite of loss of memory)
- Users may abort the program: need to “rollback changes”
  - Need to *log* what happened
- Many users executing concurrently

And all this with… Scalability and/or Performance!!

A Note: ACID is contentious!

- Many debates over ACID, both **historically** and **currently**
- Some “NoSQL” DBs relax ACID
- In turn, now “NewSQL” reintroduces ACID compliance to NoSQL-style DBs…

Atomicity and Durability via Logging

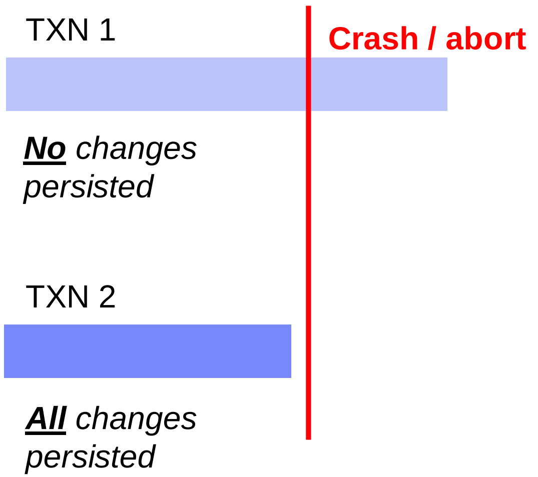


Goal for this lecture: Ensuring Atomicity & Durability

ACI**D**

- Atomicity:
  - TXNs should either happen completely or not at all
  - If abort / crash during TXN, *no* effects should be seen

- Durability:
  - If DB stops running, changes due to completed TXNs should all persist
  - *Just store on stable disk*



We'll focus on how to accomplish atomicity (via logging)

Basic Idea: (Physical) Logging

Idea:

- Log consists of an ordered list of update records
- Log record contains UNDO information for every update!

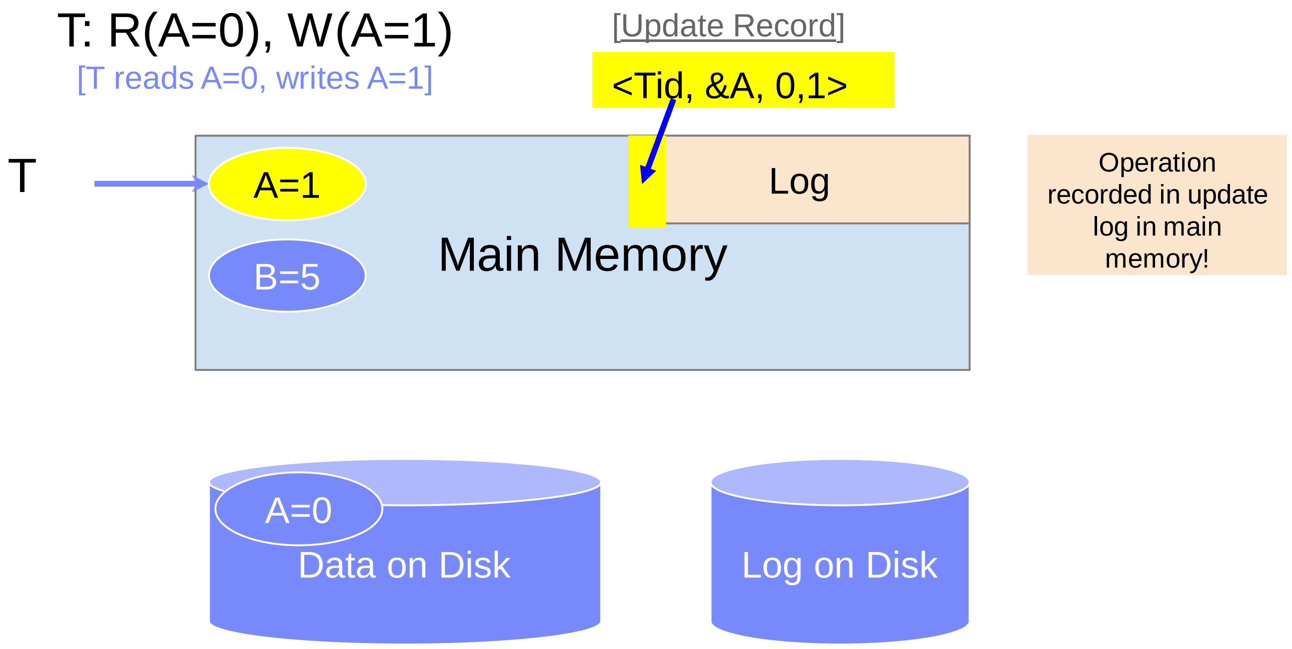
  <TransactionID, location, old data, new data>

What DB does?

- Owns the log “service” for all applications/transactions.
- Transparent to application or transaction
- Sequential writes to log, can **flush** — force writes to disk

This is sufficient to UNDO any transaction!

A picture of logging



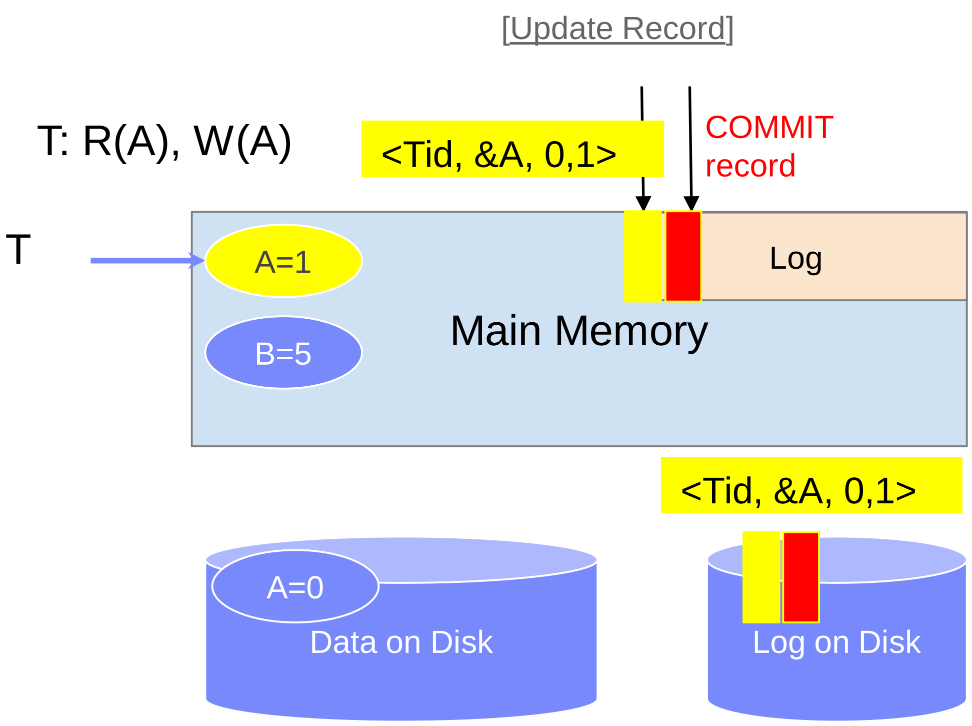
Why do we need logging for atomicity?

- Couldn't we just write TXN to disk **only** once whole TXN complete?
  - Then, if abort / crash and TXN not complete, it has no effect- atomicity!
  - *With unlimited memory and time, this could work…*

- However, we **need to log partial results of TXNs** because of:
  - Memory constraints (enough space for full TXN??)
  - Time constraints (what if one TXN takes very long?)

We need to write partial results to disk!
…And so we need a **log** to be able to ***undo*** these partial results!

Write-ahead Logging (WAL) Commit Protocol

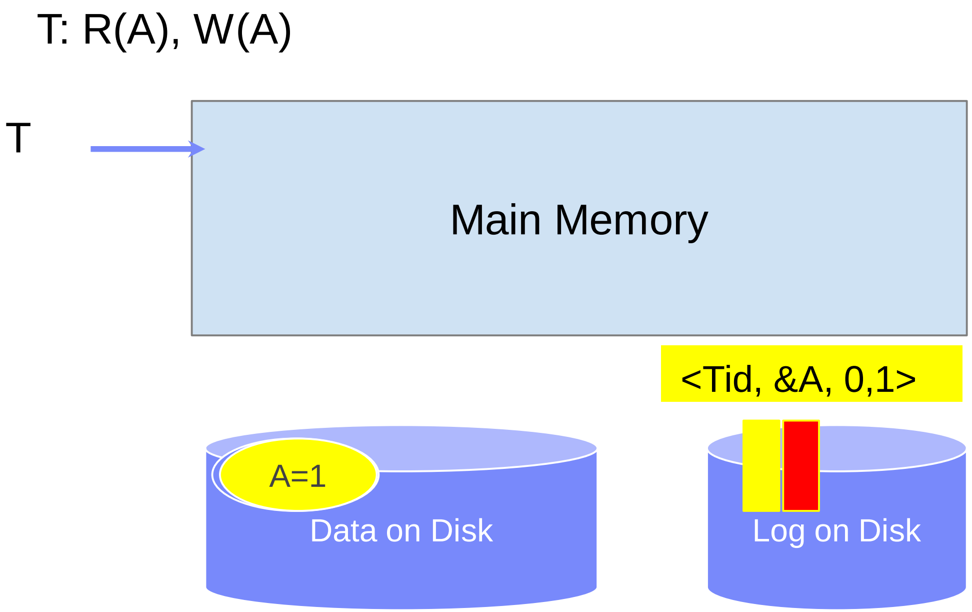


Commit after we've written log to disk but before we've written data to disk…

OK, Commit!

If we crash now, is T durable?

Write-ahead Logging (WAL) Commit Protocol



Commit after we've written log to disk but before we've written data to disk… this is WAL!

OK, Commit!

If we crash now, is T durable?

USE THE LOG!

Write-Ahead Logging (WAL)

<u>Algorithm</u>: WAL

For each record update, write <u>Update Record</u> into LOG

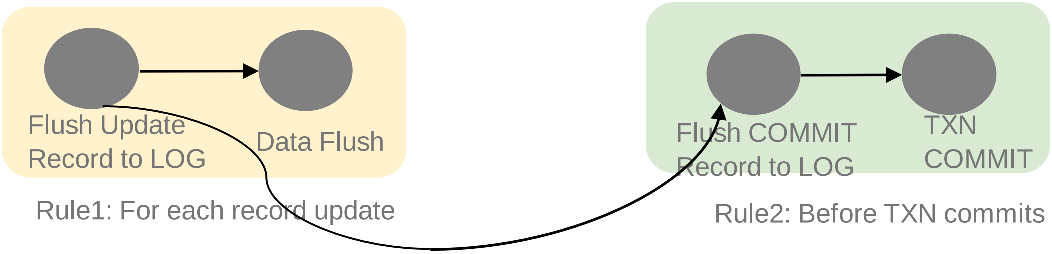
Follow two Flush rules for LOG

- **Rule1**: Flush <u>Update Record</u> *into LOG before* corresponding data page goes to storage
- **Rule2**: Before TXN commits,
  - Flush all <u>Update Records</u> to LOG
  - Flush <u>COMMIT Record</u> to LOG

→ **<u>Durability</u>**

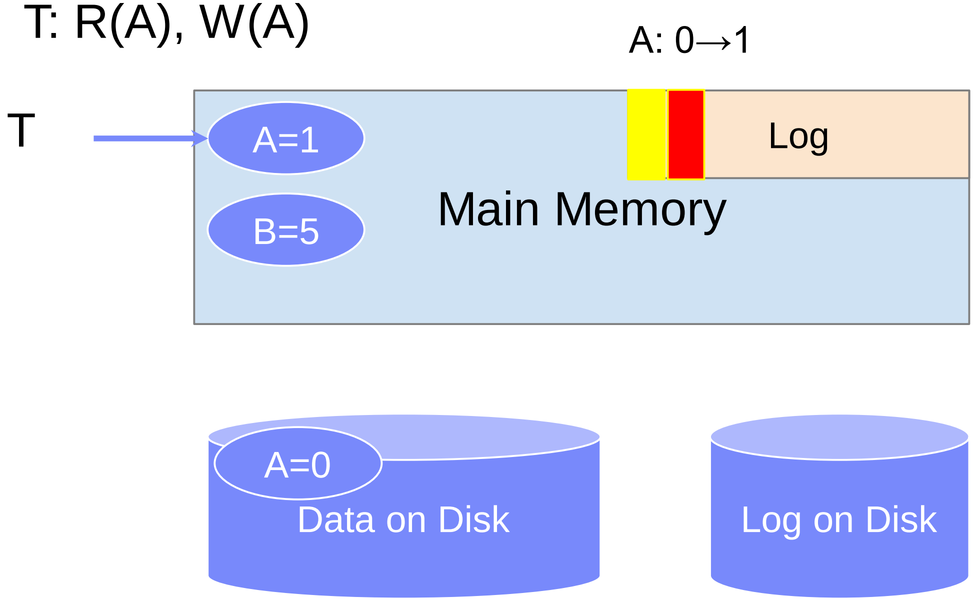
→ **<u>Atomicity</u>**

Transaction is committed *once COMMIT record is on stable storage*



Incorrect Commit Protocol #1

T: R(A), W(A)



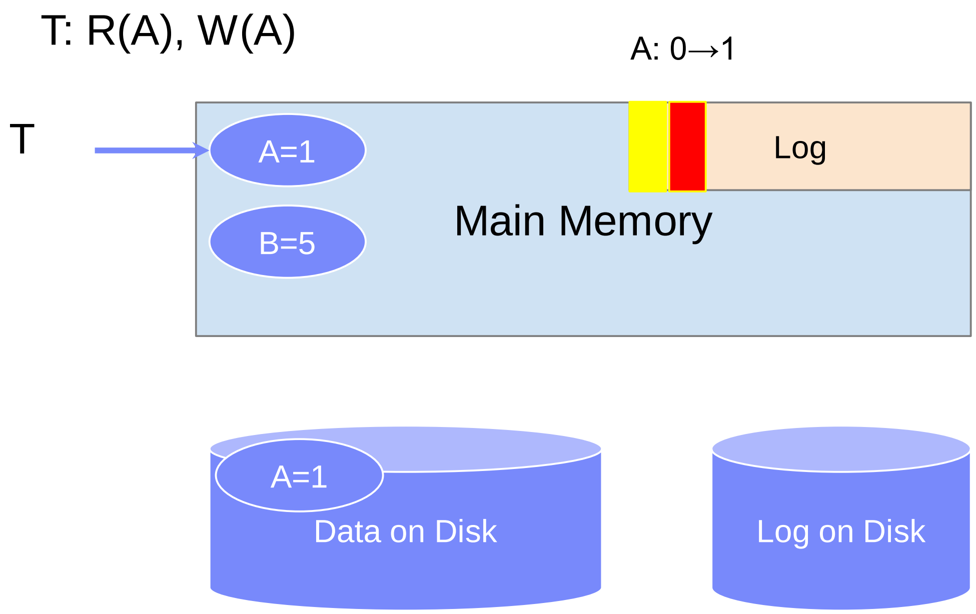
Let's try committing *before* we've written either data or log to disk…

OK, Commit!

If we crash now, is T durable?

Lost T's update!

Incorrect Commit Protocol #2



Let's try committing *after* we've written data but *before* we've written log to disk…

OK, Commit!

If we crash now, is T durable? Yes! Except…

How do we know whether T was committed??

Bank interest example: full run

Money

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance ($)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...		
30108		-110
40008		110
50002		22

WAL (@4:29 am day+1)

T-Monthly-423	**START TRANSACTION**		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	**COMMIT**		

Update Records

Commit Record

'T-Monthly-423'

Monthly Interest 10%

4:28 am Starts run on 10M bank accounts

Takes 24 hours to run

```
START TRANSACTION
        UPDATE Money
        SET Amt = Amt * 1.10
COMMIT
```

Bank interest example: with crash

Money

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...	...	
30108		-100
40008		100
50002		20

Money (@10:45 am)

Account	....	Balance ($)	
3001		550	??
4001		110	
5001		22	
6001		66	
3002		88	
4002		-200	??
5002		320	??
...			
30108		-110	
40008		110	
50002		22	??

WAL log (@10:29 am)

T-Monthly-423	**START TRANSACTION**		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352

‘T-Monthly-423’

Monthly Interest 10%
4:28 am Starts run on 10M bank accounts
Takes 24 hours to run
Network outage at 10:29 am,
System access at 10:45 am

Did T-Monthly-423 complete?
Which tuples are bad?

Case1: T-Monthly-423 was crashed
Case2: T-Monthly-423 completed. 4002 deposited 20$ at 10:45 am

Bank interest example: with recovery

Money (@10:45 am)

Account	....	Balance ($)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-200
5002		320
...		
30108		-110
40008		110
50002		22

Money (after recovery)

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

WAL log (@10:29 am)

T-Monthly-423	**START TRANSACTION**		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22

[redacted]

System recovery (after 10:45 am)

1. Rollback uncommitted transactions
   - Restore old values from WALlog (if any)
   - Notify developers about aborted txn
1. Redo Recent transactions (w/ new values)
2. Back in business; Redo (any pending) transactions

A word on performance

- Question: why is a WAL a good idea?
- Answer: updates to WAL are in sequential order!

- Recall: sequential writes are very important both for flash and magnetic disk
  - In a couple of lectures we will understand why

An example of why sequential writes matter

Money

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...	...	
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance ($)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...		
30108		-110
40008		110
50002		22

WAL (@4:29 am day+1)

T-Monthly-423	**START TRANSACTION**		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	**COMMIT**		

Cost to update all data

10M bank accounts → 10M individual random writes? (worst case)

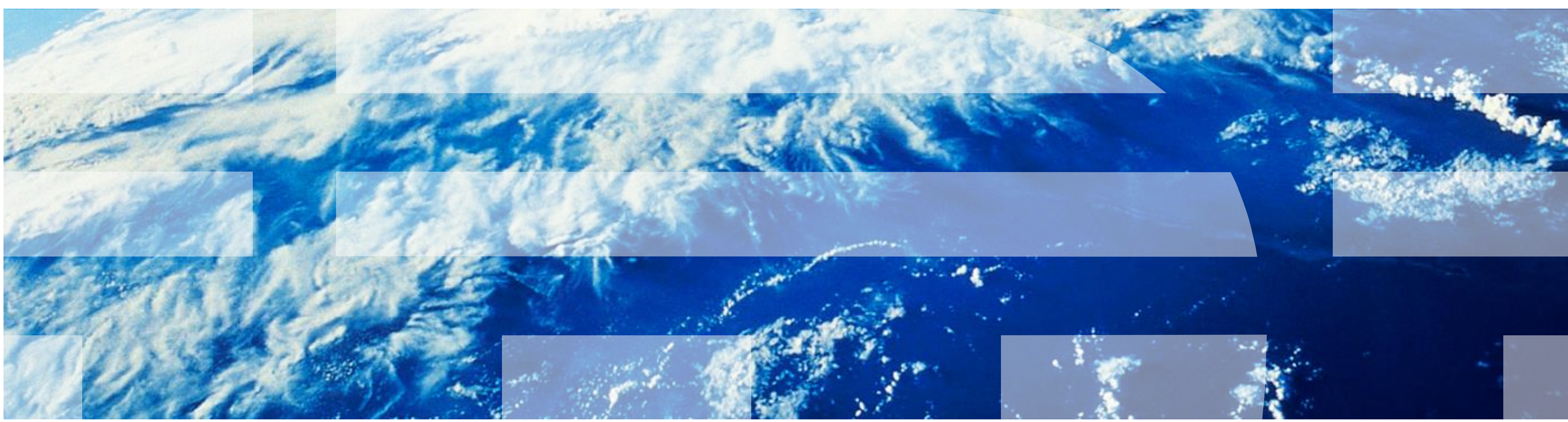
(@10 ms per write for magnetic disk, that's 100,000 secs)

Speedup for commit

100,000 secs vs 1 sec when written sequentially!!!

Borrowed from Shiva Shivakumar

Concurrency and Locking for Transactions



Back to our bank example

Money

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance ($)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...		
30108		-110
40008		110
50002		22

<u>'T-Monthly-423'</u>

Monthly Interest 10%
4:28 am Starts run on 10M bank accounts
Takes 24 hours to run

```
UPDATE Money
SET Balance = Balance * 1.1
```

<u>Other Transactions</u>

10:02 am Acct 3001: Wants 600$
11:45 am Acct 5002: Wire for 1000$
. . . . .
. . . . .
2:02 pm Acct 3001: Debit card for $12.37

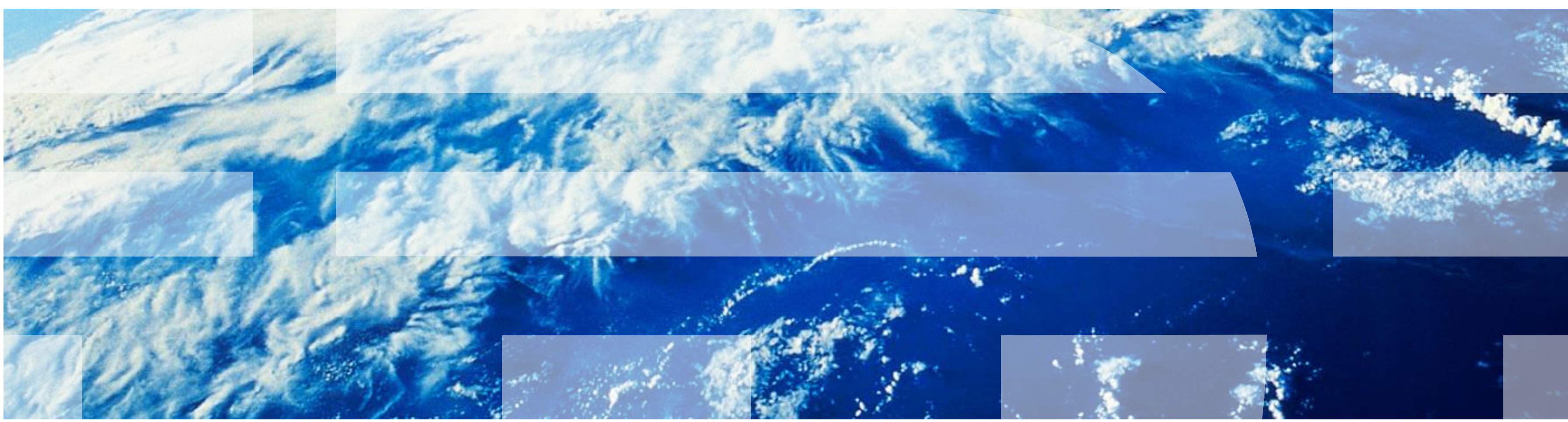
Q: How do I not wait for a day to access my $$$s?

Big idea: locks

- Intuition
  - ”Lock” each record for shortest time possible

- Key questions
  - Which records?
  - For how long?
  - What is the algorithm for holding them?

Concurrency, Scheduling and Anomalies



Concurrency: Isolation & Consistency

- DB is responsible for concurrency so that…

Isolation is maintained: Users must be able to execute each TXN as if they were the only user

AC**I**D

Consistency is maintained: TXNs must leave the DB in consistent state

A**C**ID

Example- consider two TXNs:

```
T1: START TRANSACTION
        UPDATE Accounts
        SET Amt = Amt + 100
        WHERE Name = ‘A’

        UPDATE Accounts
        SET Amt = Amt - 100
        WHERE Name = ‘B’
COMMIT
```

```
T2: START TRANSACTION
        UPDATE Accounts
        SET Amt = Amt * 1.06
COMMIT
```

T1 transfers $100 from B’s account to A’s account

T2 credits both accounts with a 6% interest payment

Note:

1. DB does not care if T1 —> T2 or T2 —> T1 (which TXN executes first)
2. If developer does, what can they do? (Put T1 and T2 inside 1 TXN)

Example

T_1 A += 100 B -= 100

T1 transfers $100 from B's account to A's account

T_2 A *= 1.06 B *= 1.06

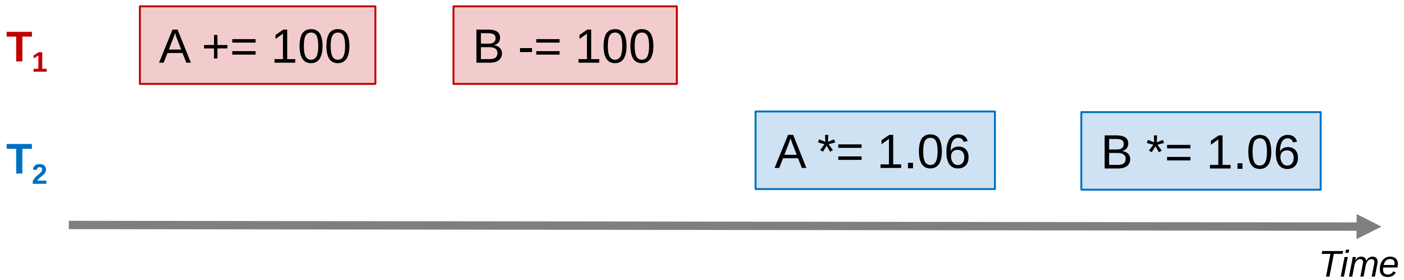
T2 credits both accounts with a 6% interest payment

Goal for scheduling transactions:

- Interleave transactions to boost performance
- Data stays in a good state after commits and/or aborts (ACID)

Example- consider two TXNs:

We can look at the TXNs in a timeline view- serial execution:

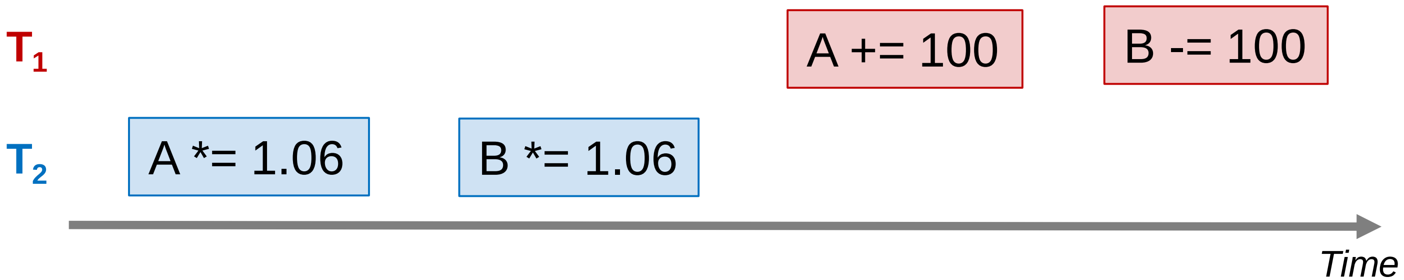


T1 transfers $100 from B's account to A's account

T2 credits both accounts with a 6% interest payment

Example- consider two TXNs:

The TXNs could occur in either order… DB allows!

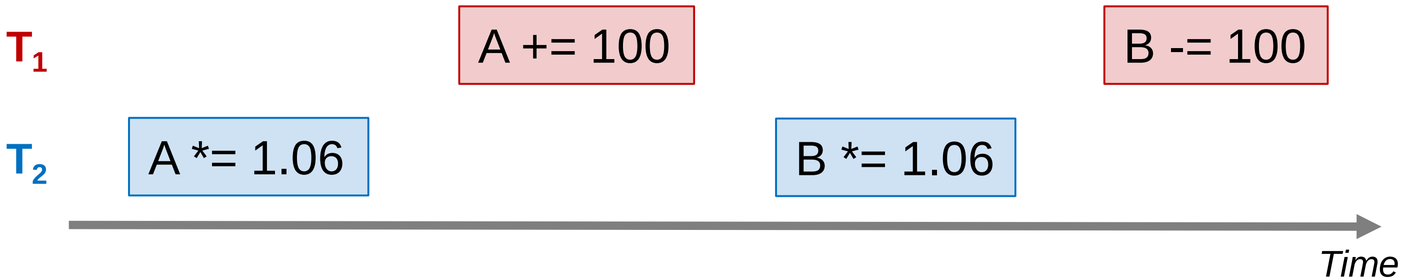


T2 credits both accounts with a 6% interest payment

T1 transfers $100 from B's account to A's account

Example- consider two TXNs:

The DB can also **interleave** the TXNs



T2 credits A's account with 6% interest payment, then T1 transfers $100 to A's account…

T2 credits B's account with a 6% interest payment, then T1 transfers $100 from B's account…

Interleaving & Isolation

- The DB has freedom to interleave TXNs
- However, it must pick an interleaving or schedule such that isolation and consistency are maintained
- ⇒ Must be *as if* the TXNs had executed serially!

"With great power comes great responsibility"

A**C****I**D

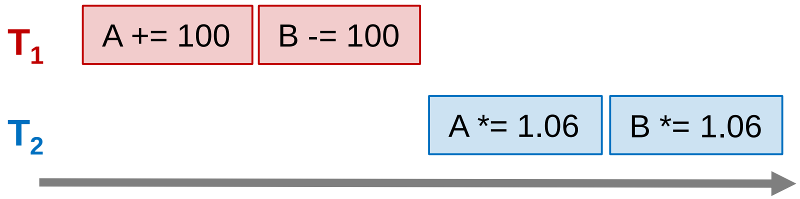
DB must pick a schedule which maintains isolation & consistency

Scheduling examples

Starting Balance

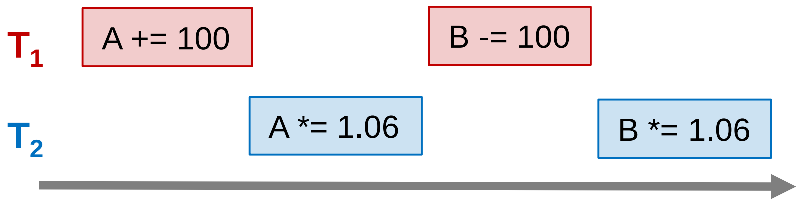
A	B
$50	$200

Serial schedule T_1,T_2:



A	B
$159	$106

Interleaved schedule A:



A	B
$159	$106

Same result!

Scheduling examples

Starting Balance

A	B
$50	$200

Serial schedule T_1,T_2:

T_1 A += 100 | B -= 100

T_2 A *= 1.06 | B *= 1.06

A	B
$159	$106

Interleaved schedule B:

T_1 A += 100 | B -= 100

T_2 A *= 1.06 | B *= 1.06

A	B
$159	**$112**

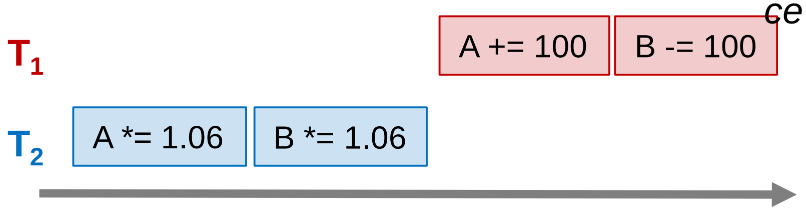
Different result than serial T_1,T_2!

Scheduling examples

Starting Balance

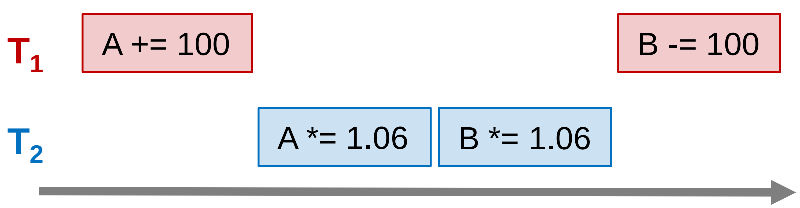
A	B
\$50	\$200

Serial schedule ***T_2,T_1:***



A	B
\$153	\$112

Interleaved schedule B:

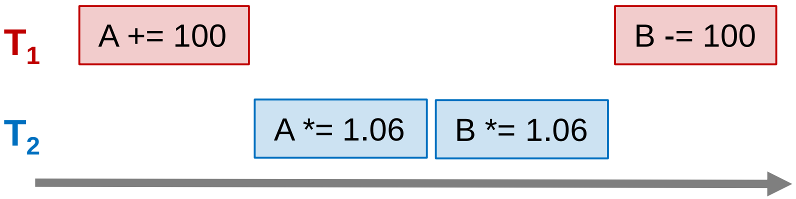


A	B
\$159	\$112

Different result than serial T_2,T_1 ALSO!

Scheduling examples

Interleaved schedule B:



This schedule is different than ***any serial order!*** We say that it is **not serializable**

Scheduling Definitions

- A **serial schedule** is one that does not interleave the actions of different transactions
- A and B are **equivalent schedules** if, ***for any database state***, the effect on DB of executing A **is identical to** the effect of executing B
- *A* **serializable schedule** is a schedule that is equivalent to ***some*** serial execution of the transactions.

The word "**some"** makes this definition powerful & tricky!

Serial Schedules

T1		A += 100	B -= 100		
T2				A *= 1.06	B*= 1.06

S1

T1				A += 100	B -= 100
T2		A *= 1.06	B *= 1.06		

S2

Interleaved Schedules

T1		A += 100		B -= 100	
T2			A *= 1.06		B*= 1.06

S3

T1			A += 100		B -= 100
T2		A *= 1.06		B *= 1.06	

S4

T1			A += 100	B -= 100	
T2		A *= 1.06			B*= 1.06

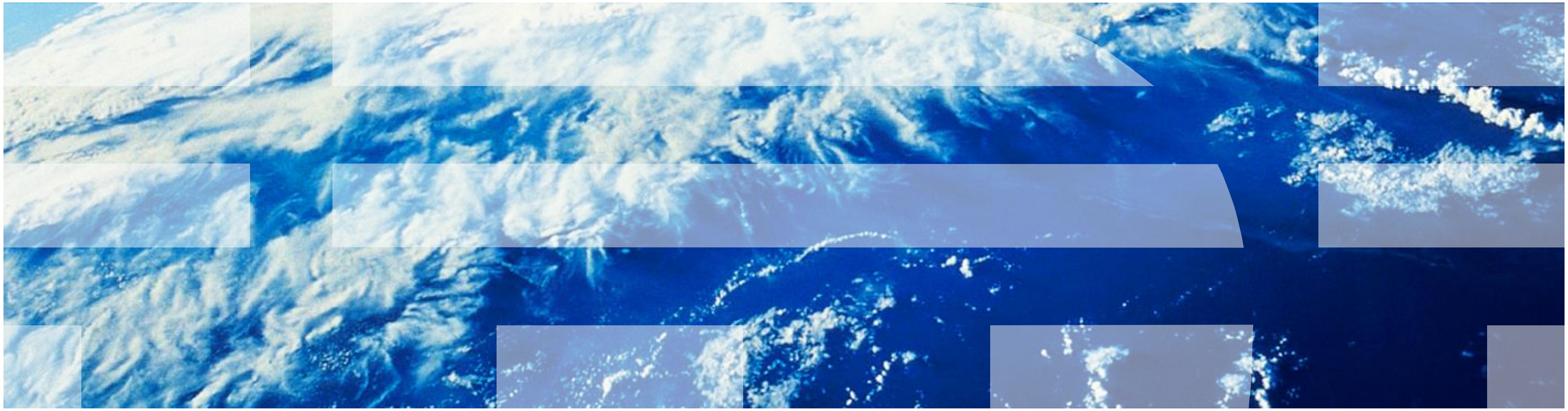
S5

T1		A += 100			B -= 100
T2			A *= 1.06	B *= 1.06	

S6

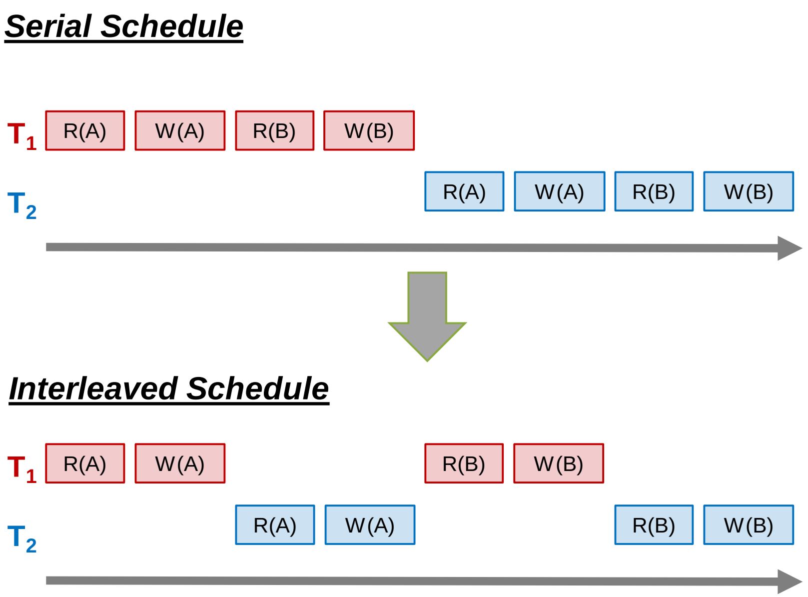
Serial Schedules	S1, S2
Serializable Schedules	S3, S4 (And S1, S2)
Equivalent Schedules	<S1, S3> <S2, S4>
Non-serializable (Bad) Schedules	S5, S6

Conflicts and Anomalies



General DB model:
Concurrency as Interleaving TXNs

Each action in the TXNs *reads a value* and then *writes one back*



For our purposes, having TXNs occur concurrently means **interleaving their component actions (R/W)**

We call the particular order of interleaving a **schedule**

Conflict Types

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write

Thus, there are three types of conflicts:

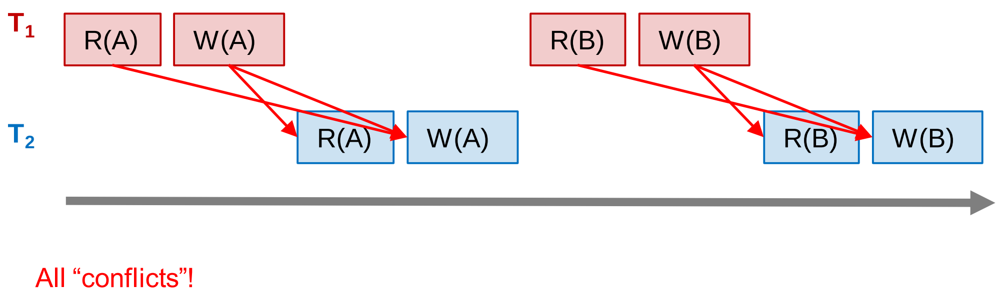
- Read-Write conflicts (RW)
- Write-Read conflicts (WR)
- Write-Write conflicts (WW)

Why no "RR Conflict"?

Note: **conflicts** happen often in many real world transactions. (E.g., two people trying to book an airline ticket)

Conflicts

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write



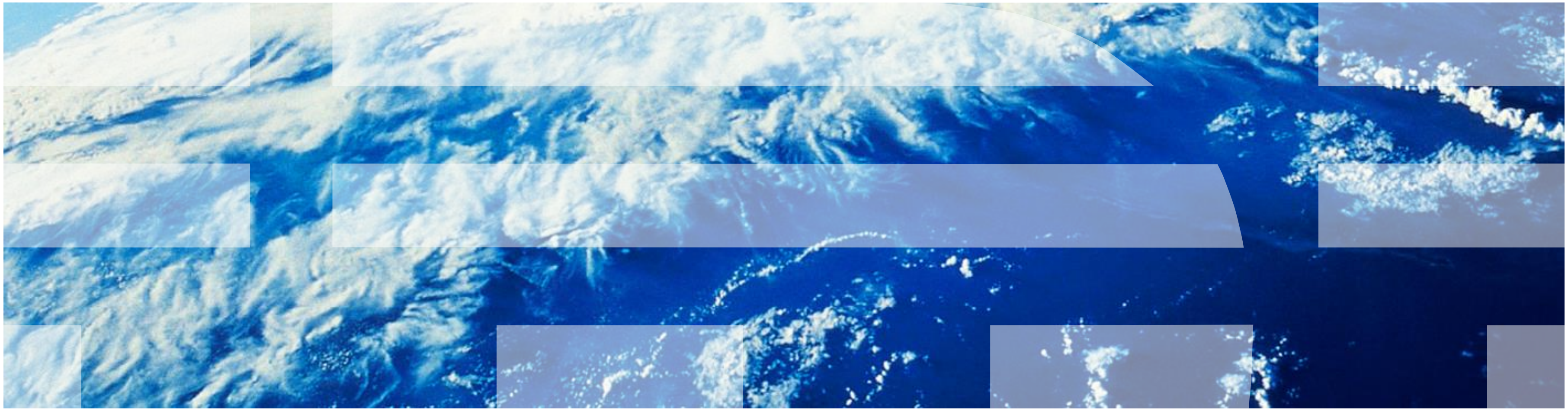
Note: Conflicts vs. Anomalies

<u>Conflicts</u> are in both "good" and "bad" schedules
(they are a property of transactions)

Goal: Avoid <u>Anomalies</u> while interleaving transactions with conflicts!

- Do not create "bad" schedules where isolation and/or consistency is broken (i.e., Anomalies)

Conflict Serializability



Conflict Serializability

Two schedules are **conflict equivalent** if:

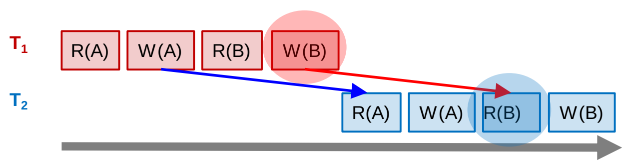
- They involve *the same actions of the same TXNs*
- Every *pair of conflicting actions* of two TXNs are *ordered in the same way*

Schedule S is **conflict serializable** if S is *conflict equivalent* to some serial schedule

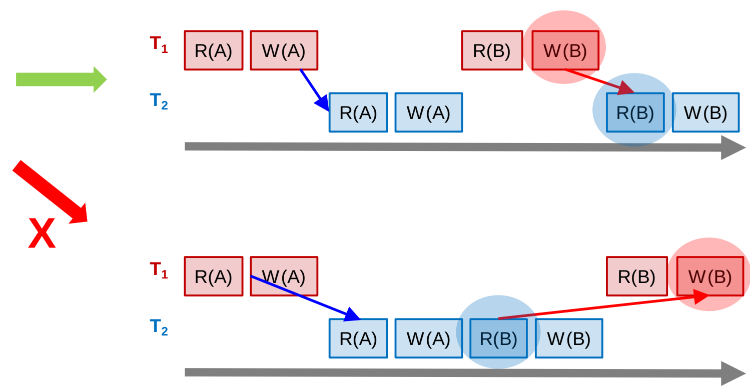
Conflict serializable ⇒ serializable
So if we have conflict serializable, we have consistency & isolation!

Example “Good” vs. “bad” schedules

Serial Schedule:



Interleaved Schedules:

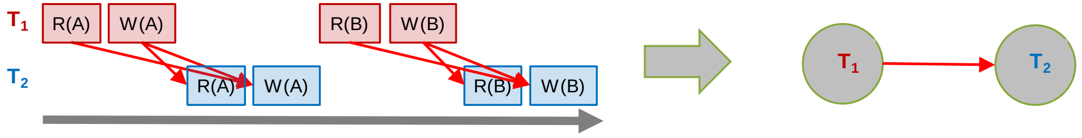


Note that in the “bad” schedule, the ***order of conflicting actions is different than the above (or any) serial schedule!***

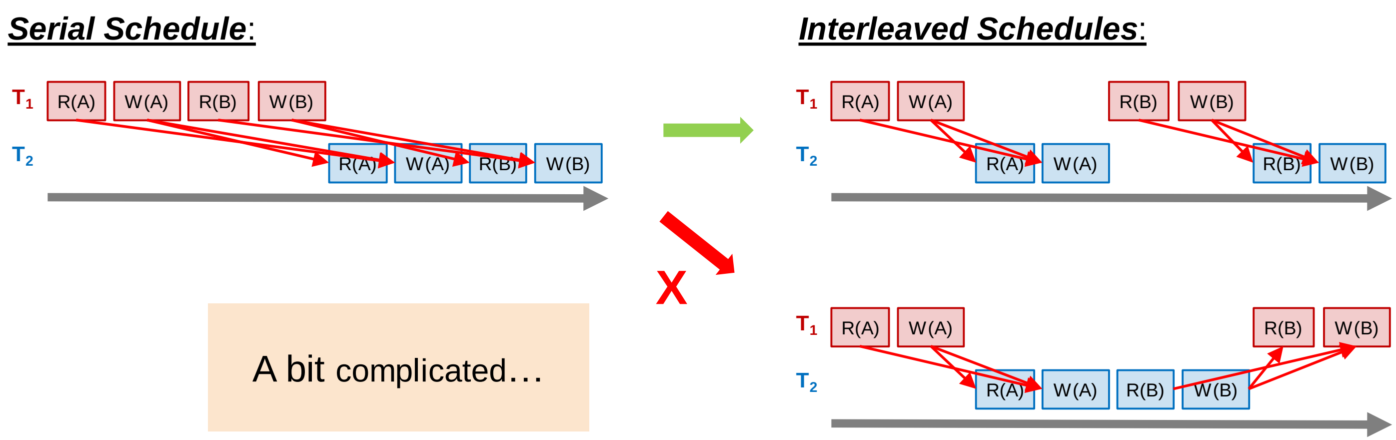
Conflict serializability provides us with an operative notion of “good” vs. “bad” schedules! “Bad” schedules create data Anomalies

The Conflict Graph

- Let's now consider looking at conflicts **at the TXN level**
- Consider a graph where the **nodes are TXNs**, and there is an edge from $T_i \rightarrow T_j$ **if any actions in T_i <u>precede and conflict with</u> any actions in T_j**

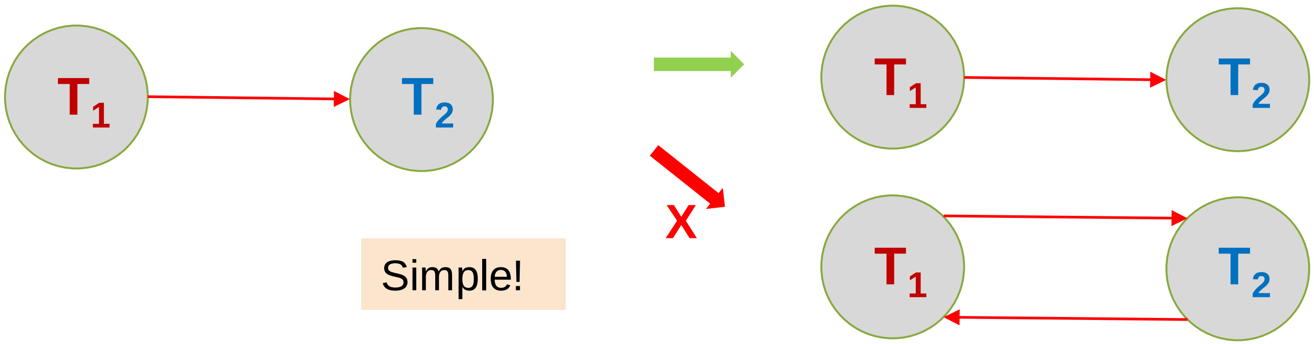


What can we say about “good” vs. “bad” conflict graphs?



What can we say about “good” vs. “bad” conflict graphs?

Serial Schedule: ***Interleaved Schedules***:



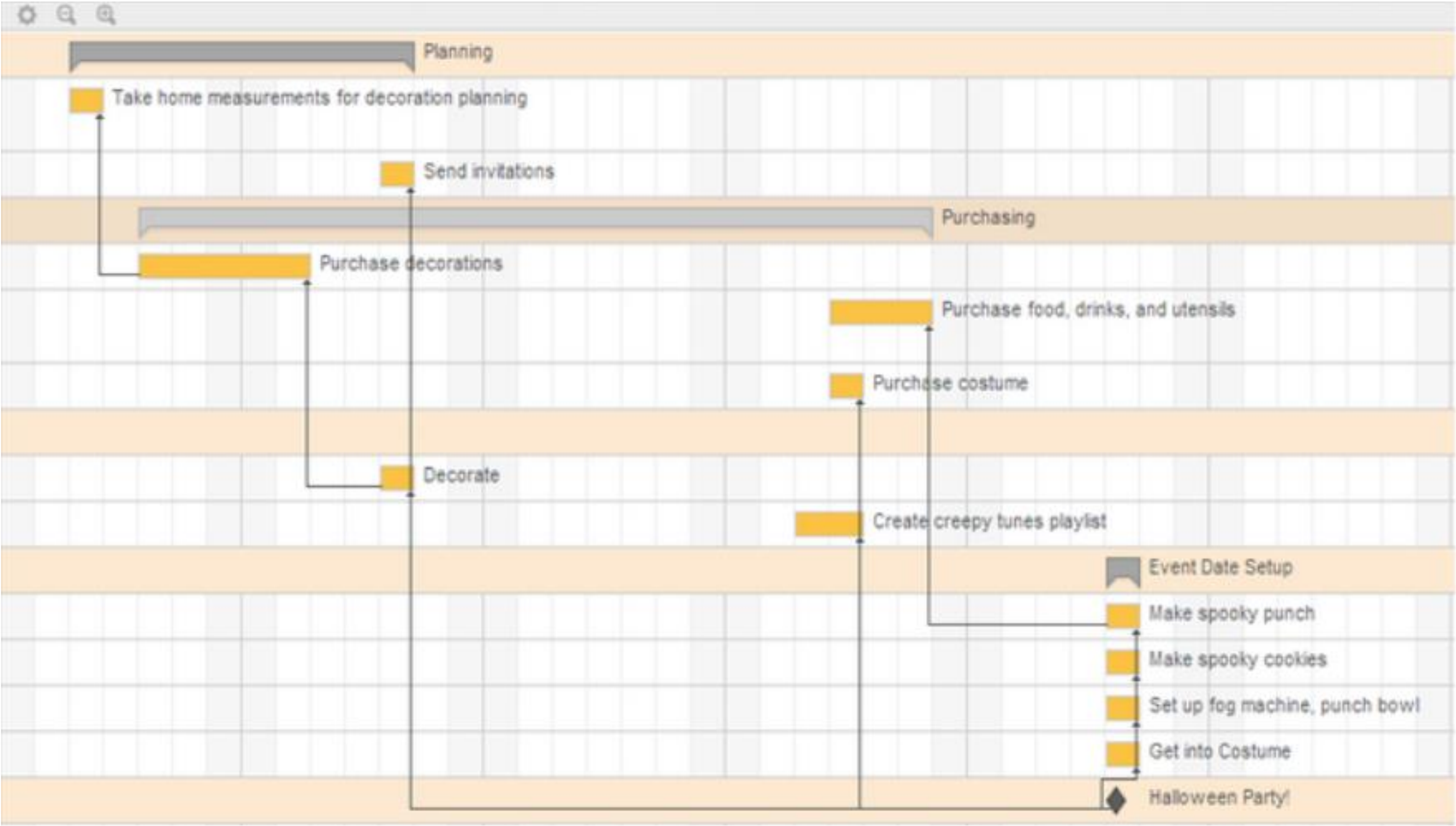
Theorem: Schedule is **conflict serializable** if and only if its conflict graph is **acyclic**

DAGs & Topological Orderings

- A **topological ordering** of a directed graph is a linear ordering of its vertices that respects all the directed edges

- A directed **acyclic** graph (DAG) always has one or more **topological orderings**
  - (And there exists a topological ordering *if and only if* there are no directed cycles)

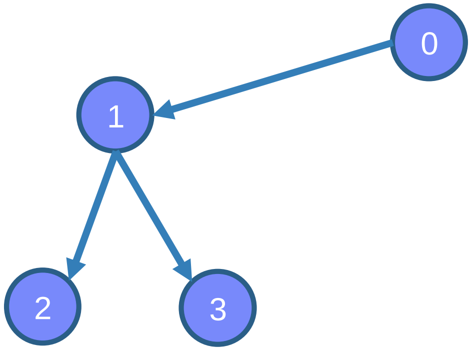
Example: Project dependencies



How would you plan?
What if there are cycles? (dependencies)

DAGs & Topological Orderings

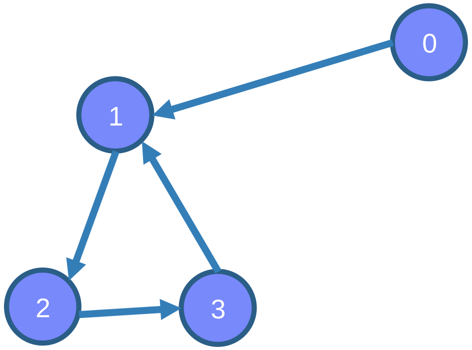
- Ex: What is one possible topological ordering here?



Ex: 0, 1, 2, 3 (or: 0, 1, 3, 2)

DAGs & Topological Orderings

- Ex: What is one possible topological ordering here?



There is none!

Connection to conflict serializability

- In the conflict graph, a topological ordering of nodes corresponds to **a serial ordering of TXNs**
- Thus an **acyclic** conflict graph → conflict serializable!

Theorem: Schedule is **conflict serializable** if and only if its conflict graph is **acyclic**

Example with 5 transactions

Schedule S1

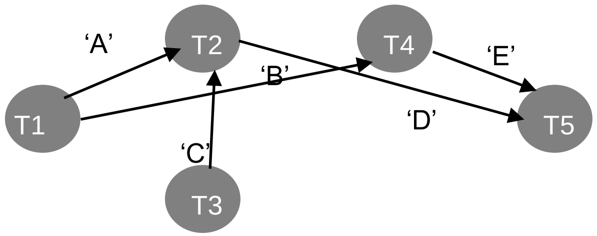
Good or Bad schedule?
Conflict serializable?

w1(A)	r2(A)	w1(B)	w3(C)	r2(C)	r4(B)	w2(D)	w4(E)	r5(D)	w5(E)

Step1
Find conflicts
(RW, WW, WR)

T1	w1(A)		w1(B)							
T2		r2(A)			R2(C)		w2(D)			
T3				w3(C)						
T4						r4(B)		w4(E)		
T5									r5(D)	w5(E)

Step2
Build Conflict graph
Acyclic?



Acyclic
⇒ Conflict serializable!
⇒ Serializable

S2

Step3
Example serial schedule
Conflict Equiv to S1

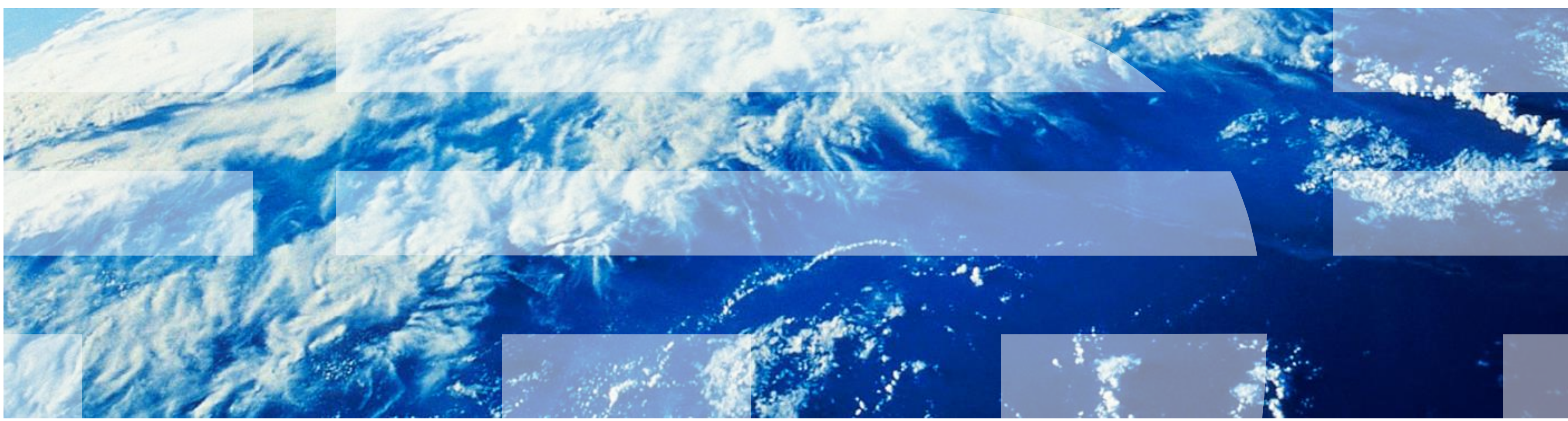
T3	T1	T4	T4	T5	T2	T2	T5	T5
w3(C)	w1(B)	r4(B)	w4(E)	r2(A)	r2(C)	w2(D)	r5(D)	w5(E)

Summary

- Concurrency achieved by **interleaving TXNs** such that **isolation** & **consistency** are maintained
  - We formalized a notion of **serializability** that captured such a "good" interleaving schedule

- We defined **conflict serializability**

2PL: A Simple Locking Algorithm



Strict Two-Phase Locking (2PL)

- Algorithm: *strict two-phase locking* - as a way to deal with concurrency
  - Guarantees conflict serializability
  - (if it completes- see upcoming…)

- Also (*conceptually*) straightforward to implement, and transparent to the user!

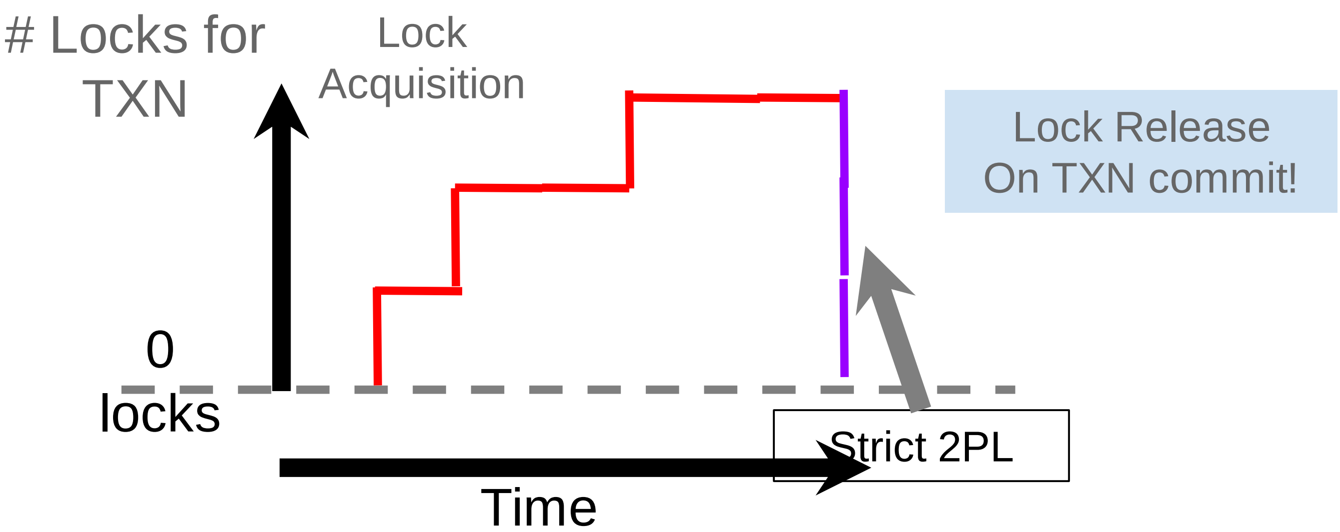
Strict Two-phase Locking (2PL) Protocol

TXNs obtain:

- An **X (*exclusive*) lock** on object before **writing**.
  - If a TXN holds, no other TXN can get a lock (S or X) on that object.
- An **S (*shared*) lock** on object before **reading**
  - If a TXN holds, no other TXN can get *an X lock* on that object
- All locks held by a TXN are released when TXN completes.

Note: Terminology here- "exclusive", "shared"- meant to be intuitive- no tricks!

Picture of 2-Phase Locking (2PL)



2PL: A transaction can not request additional locks once it releases any locks. Thus, there is a “growing phase” followed by a “shrinking phase”.

Strict 2PL: Release locks only at COMMIT (COMMIT Record flushed) or ABORT

Strict 2PL

If a schedule follows strict 2PL, it is **conflict serializable**…

- …and thus serializable
- …and we get isolation & consistency!

Popular implementation

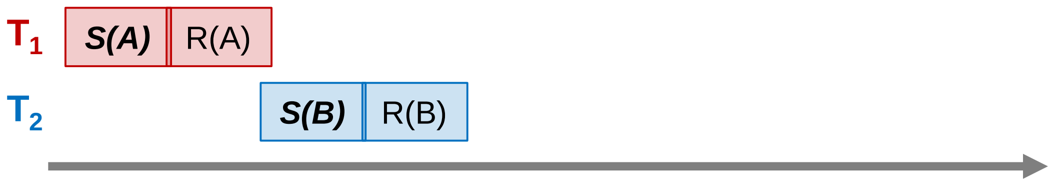
- Simple !
- Produces subset of *all* conflict serializable schedules
- There are MANY more complex LOCKING schemes with better performance. (See CS Database classes)

- One key, subtle problem (next)

Deadlock Detection

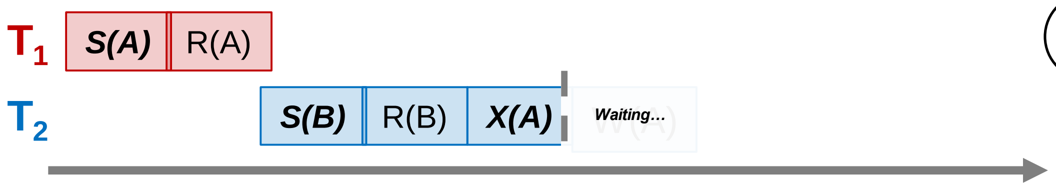
First, T_1 requests a shared lock on A to read from it

Deadlock Detection



Next, T_2 requests a shared lock on B to read from it

Deadlock Detection: Example

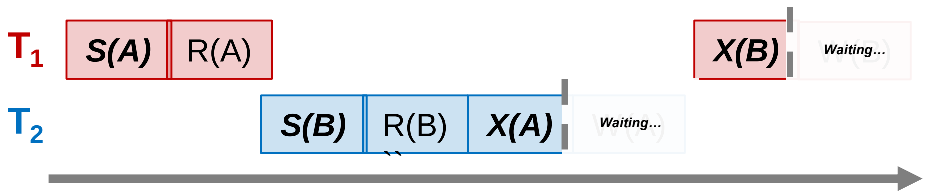


Waits-for graph:

T_2 then requests an exclusive lock on A to write to it- **now T_2 is waiting on T_1…**

Waits-For graph: Track which Transactions are waiting
<u>IMPORTANT</u>: WAITS-FOR graph different than CONFLICT graph we learnt earlier !

Deadlock Detection: Example



Waits-for graph:

Cycle = DEADLOCK

Finally, T_1 requests an exclusive lock on B to write to it- **now T_1 is waiting on T_2… DEADLOCK!**

Deadlocks

Deadlock: Cycle of transactions waiting for locks to be released by each other.

Two ways of dealing with deadlocks:

- Deadlock prevention
- Deadlock detection

Deadlock Detection

Create the **waits-for graph**:

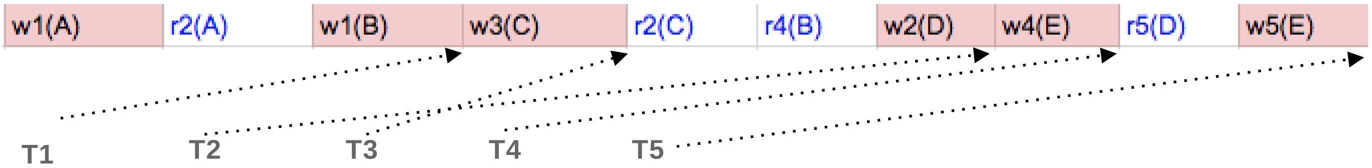
- Nodes are transactions
- There is an edge from $T_i \rightarrow T_j$ if T_i is *waiting for T_j to release a lock*

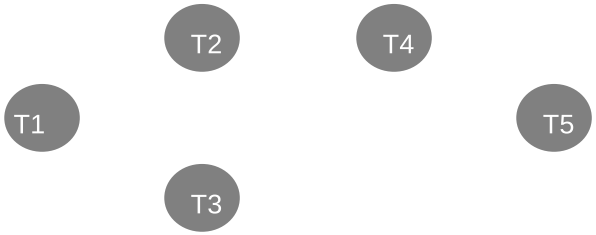
Periodically check for (***and break***) cycles in the waits-for graph

Example with 5 Transactions (2PL)

Schedule S1

Execute with 2PL





Waits- For Graph

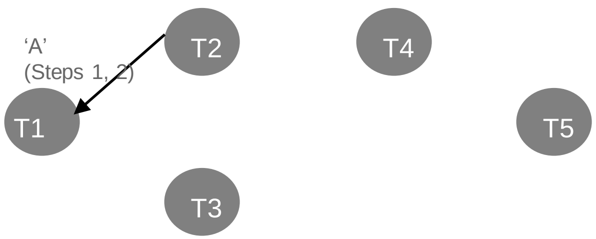
Example with 5 Transactions (2PL)

Schedule S1

w1(A)	r2(A)	w1(B)	w3(C)	r2(C)	r4(B)	w2(D)	w4(E)	r5(D)	w5(E)

Execute with 2PL

	T1	T2	T3	T4	T5
Step 0	X (A) w1(A)				
Step 1		Req S(A)			
Step 2	X (B) w1(B) Unl B, A				
Step 3		Get S(A) r2(A)			
Step 4			X (C) w3(C) Unl C		
Step 5		S(C) r2(C)			
Step 6				S(B) r4(B)	
Step 7		X(D) w2(D) Unl A, C, D			
Step 8				X(E) w4(E) Unl B,E	
Step 9					S (D) r5(D)
Step 10					X (E) w5(E) Unl D, E



Summary

Locking allows only conflict serializable schedules

- If the schedule completes… (it may deadlock!)